Types

Deian Stefan

(adopted from my & Edward Yang’s CSE242 slides)
Today

- General discussion of types
- Type inference
- Type polymorphism
What is a type?

• Examples of types:
  ➤ Integer
  ➤ [Char]
  ➤ Either (Either Char Int) Bool

• Working, informal definition: set of values
  ➤ Where does this definition break down?
A type is: a way to prevent errors

- E.g.,

```javascript
const y = 1;
y + "w00t";
```

- E.g.,

```javascript
function apply(f, x) {
    return f(x);
}
```
A type is: a way to prevent errors

- E.g.,
  --| Function must be applied to 2 Ints
  plus :: Int -> Int -> Int
  plus a b = ...

- E.g.,
  --| Must be applied to a function and
  -- argument that that function can be applied to
  apply :: (a -> b) -> a -> b
  apply f x = f x
A type is: a way to prevent errors

- The world’s most lightweight* and widely-used formal method!
  - Prevent meaningless computations from being expressed or executed
A type is: a method of organization & documentation

• E.g., consider abstract data type for sets

```haskell
data Set k = ...
empty :: Set k
insert :: k -> Set k -> Set k
delete :: k -> Set k -> Set k
member :: k -> Set k -> Bool
```

• E.g., consider type for reading a file

```haskell
readFile :: FilePath -> IO String
```
A type is: a hint to the compiler

- E.g., what should obj.prop1 be compiled down to?
Who enforces types?

- Consider, for example: `arr[200]`
  - What happens in JavaScript if `arr` is `null`?
  - What happens in C/C++ if `arr` is of size 10?
  - What happens in Haskell if `arr` is not an array?
Who enforces types?

• This is language dependent...

  ➤ The compiler at compile time

  ➤ The runtime system at run-time

  ➤ The hardware at run-time
What are the tradeoffs of each?

<table>
<thead>
<tr>
<th></th>
<th>Compile-time</th>
<th>Run-time checks</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pro</td>
<td>No runtime overhead</td>
<td>Permissive</td>
<td>Super fast</td>
</tr>
<tr>
<td>Con</td>
<td>Over approximates</td>
<td>Runtime overhead</td>
<td>Catch bugs late</td>
</tr>
</tbody>
</table>
Compile-time is the best! (Is it?)
The cost of compile-time checking

• Sometimes you give up expressivity

```javascript
function f(x) {
    return x < 10 ? x : x();
}
```

➤ More advanced type systems can “type” this function (dependent types); at what cost?

• Why is this fundamental? A: static analysis approximates — it has to work for every run of the program
Why do we check types? Safety!

- **Def:** A language is **type safe** if no program is allowed to violate its type distinctions

  ➤ Is Haskell type safe? **A: yes, B: no**

  ➤ Is JavaScript type safe? **A: yes, B: no**

  ➤ Is C/C++ type safe? **A: yes, B: no**

- What language features make it hard to guarantee type safety? **A: raw pointer/memory access, casts, etc.**
Today

• General discussion of types ✓
• Type inference ✓
• Type polymorphism
<2 min interlude>
Type inference

• What’s the difference between type checking and type inference?

➤ E.g.,

```java
int f(int x) {
    return x + 1;
}
```

➤ Type checking: checks that x is actually used as an int

➤ Type inference: based usage infers that x is an int
Why study type inference?

- Reduces syntactic overhead of expressive types
- Guaranteed to produce the most general type
- One of the most important language innovations
  - Even C++ has type inference now!
- Good example of a flow-insensitive static analysis alg
What we’re going to look at

Hindley-Milner type inference for uHaskell!
Hindley-Milner type inference

• [1958] Curry and Feys invented type inference algorithm for the simply typed $\lambda$ calculus

• [1969] Hindley extended algorithm to richer language and proved it always produced most general type

• [1978] Milner developed Algorithm W

• [1982] Damas proved the algorithm was complete
Hindley-Milner type inference

1. Parse the program

2. Assign type variables to all nodes

3. Generate constraints between type variables

4. Solve constraints (via unification)

5. Read out types of top-level declarations
• Declarations:  \( d ::= \text{name p} = e \)

• Patterns:  \( p ::= \text{id} \mid (p, p) \mid p:p \mid [] \)

• Expressions  \( e ::= n \mid \text{True} \mid \text{False} \mid [] \mid \text{id} \mid (e) \mid e \oplus e \mid e e \mid (e,e) \mid \text{if } e \text{ then } e \text{ else } e \)

• Types:  \( \tau ::= \tau \rightarrow \tau \mid [\tau] \mid (\tau, \tau) \mid \text{Bool} \mid \text{Int} \)
Type inference by example

1. Basic idea

2. Polymorphism

3. Data types

4. Type error: cannot unify

5. Type error: occurs check
Ex1. Basic idea

• Example: \( f \ x = 2 + x \)

• Goal: What is the type of \( f \)? Let’s do it informally:

  ➤ \( 2 :: \text{Int} \)

  ➤ \( (+) :: \text{Int} \to \text{Int} \to \text{Int} \)

  ➤ We are applying \((+)\) to \(x\), we need \(x :: \text{Int}\)

  ➤ Thus: \( f \ x = 2 + x :: \text{Int} \to \text{Int} \to \text{Int} \)
• Step 1: parse program to construct parse tree
• Step 2: assign type variables to nodes
• Step 3: add constraints
• Step 4: solve constraints via unification
Step 5: read out type
Ex1. Basic idea

• Step 1: parse program to construct parse tree

\[
\begin{align*}
\text{f } \text{x} &= 2 + \text{x} \\
\text{= } \\
\text{= }
\end{align*}
\]
Ex1. Basic idea

- Step 2: assign type variables to nodes

\[
\begin{align*}
f\ x &= 2 + x \\
     &= (+) 2 \ x \\
     &= ((+) 2) \ x
\end{align*}
\]
Ex1. Basic idea

• Step 3: add constraints

\[ f \ x = 2 + x \]
\[ = (+) 2 \ x \]
\[ = ((+) 2) \ x \]
Generating constraints

- Lambda abstraction ($\lambda x. e$)

  \[\tau_0 = \tau_1 \rightarrow \tau_2\]

\[
\begin{array}{c}
\lambda \\
\tau_0 \\
\tau_1 \\
\tau_2 \\
x \\
e
\end{array}
\]
Generating constraints

• Lambda abstraction ($\lambda x. e$)

\[ \tau_0 = \tau_1 \rightarrow \tau_2 \]
Generating constraints

• Function declaration \((f \ x = e)\)

\[
\tau_0 = \tau_1 - \tau_2 \\
\text{Fun} \leftarrow \tau_0 \to \tau_1 \to \tau_2
\]

- \(\tau_0 = \)
Generating constraints

- Function declaration \( f \ x = e \)

\[ \tau_0 = \tau_1 \rightarrow \tau_2 \]
Generating constraints

• Function application (f x)

\[\tau_0 = \tau_1 - \tau_2\]

\[\text{Diagram: } \tau_0 \rightarrow f \rightarrow \tau_2 \rightarrow x \rightarrow \tau_1\]
Generating constraints

- Function application (f x)
  \[ \tau_0 = \tau_1 \rightarrow \tau_2 \]
Ex1. Basic idea

• Step 4: solve constraints via unification

\[ \tau_0 = \tau_1 \rightarrow \tau_6 \]
\[ \tau_2 = \tau_3 \rightarrow \tau_4 \]
\[ \tau_4 = \tau_1 \rightarrow \tau_6 \]
\[ \tau_2 = \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ \tau_3 = \text{Int} \]
Ex1. Basic idea

- Step 5: read out type

\[ \tau_0 = \text{Int} \to \text{Int} \]
\[ \tau_1 = \text{Int} \]
\[ \tau_2 = \text{Int} \to \text{Int} \to \text{Int} \]
\[ \tau_3 = \text{Int} \]
\[ \tau_4 = \text{Int} \to \text{Int} \]
\[ \tau_6 = \text{Int} \]

\[
\text{f} :: \tau_0
\]
\[
\text{f} :: \text{Int} \to \text{Int} \to \text{Int}
\]
Hindley-Milner type inference

1. Parse the program

2. Assign type variables to all nodes

3. Generate constraints

4. Solve constraints (via unification)

5. Read out types of top-level declarations
Today

- General discussion of types ✓
- Type inference ✓
- Type polymorphism
Ex2. Polymorphism

• Example: $f \circ g = g \circ 2$
Ex2. Polymorphism

• Example: \( f \ g = g \ 2 \)

\( \tau_0 = \tau_1 \rightarrow \tau_4 \)
\( \tau_1 = \tau_3 \rightarrow \tau_4 \)
\( \tau_3 = \text{Int} \)
Ex2. Polymorphism

- Example: \( f \ g = g \ 2 \)

\( \tau_0 = (\tau_3 \to \tau_4) \to \tau_4 \)

\( \tau_1 = \tau_3 \to \tau_4 \)

\( \tau_3 = \text{Int} \)
Ex2. Polymorphism

• Example: \( f \circ g = g \circ 2 \)

\[
\begin{align*}
\tau_0 &= (\text{Int} \to \tau_4) \to \tau_4 \\
\tau_1 &= \text{Int} \to \tau_4 \\
\tau_3 &= \text{Int}
\end{align*}
\]
Ex2. Polymorphism

• \( f :: (\text{Int} \to \tau_4) \to \tau_4 \) is the most general type

• What does this type mean?

• This form of polymorphism is called parametric polymorphism

• Function may have many less general types:
  
  ➤ \( f :: (\text{Int} \to \text{Int}) \to \text{Int} \)

  ➤ \( f :: (\text{Int} \to \text{Bool}) \to \text{Bool} \)
Ex2. Polymorphism

• Haskell polymorphic function
  ➤ Function $f$ is compiled into one function that works for any type

• C++ templated function
  ➤ Function $f$ is implemented $n$ different times for each unique application usage
Ex3. Data types

• Infer the type of length function:

\[ \text{len} [] = 0 \]
\[ \text{len} (x:xs) = 1 + \text{len} \; xs = (\text{+} \; 1 \; (\text{len} \; xs)) \]
Ex3. Data types

- Infer the type of length function:

\[
\text{len} \; [] = 0 \\
\text{len} \; (x:xs) = 1 + \text{len} \; xs = (+ \; 1 \; (\text{len} \; xs))
\]
Ex3. Data types

- Infer the type of length function:

\[
\text{len} \; [] = 0 \\
\text{len} \; (x:xs) = 1 + \text{len} \; xs = (+ \; 1 \; (\text{len} \; xs))
\]
Ex3. Data types

- Infer the type of length function: \( \text{len} :: [\tau_1] \to \text{Int} \)

\[
\begin{align*}
\text{len} [] &= 0 \\
\text{len} (x:xs) &= 1 + \text{len} \; xs = (+ \; 1 \; (\text{len} \; xs))
\end{align*}
\]
Infer the type of length function: \( \text{len} :: [\tau_1] \to \text{Int} \)

\[
\text{len} \; [] = 0
\]

\[
\text{len} \; (x:xs) = 1 + \text{len} \; xs = (+ 1 \; (\text{len} \; xs))
\]

➤ Infer type of each clause

➤ Combine by adding constraint: all clauses must have same type
Ex3. Data types

• What are the constraints generated by tuples?
Type inference by example

1. Basic idea ✓

2. Polymorphism ✓

3. Data types ✓

4. Type error: cannot unify

5. Type error: occurs check
Ex 4. Type errors: cannot unify

- Catch type errors by failing to unify

  ➤ Example: \( f \ x = \text{if } x \text{ then } x \text{ else } [] \)
Ex 4. Type errors: cannot unify

- Catch type errors by failing to unify

Example: \( f \ x = \text{if} \ x \ \text{then} \ x \ \text{else} \ [] \)
Ex 4. Type errors: cannot unify

- Catch type errors by failing to unify

➤ Example: \( f \ x = \text{if } x \text{ then } x \text{ else } [] \)

\[ \tau_0 = \tau_1 \rightarrow \tau_5 \]
\[ \tau_5 = \tau_1 \]
\[ \tau_1 = \tau_4 \]
\[ \tau_4 = \text{Bool} \]
\[ \tau_4 = [\tau_6] \]

\[ \tau_1 = \text{Bool} \neq \tau_4 = [\tau_6] \]
Ex5. Type error: occurs check

• Suppose we want to infer the type of $f = f \ @ f$

\[ \tau_0 = \tau_3 \]
\[ \tau_0 = \tau_0 \rightarrow \tau_3 \]
Ex5. Type error: occurs check

- Suppose we want to infer the type of \( f = f \ f \ f \)

![Diagram with type annotations]

\[ \tau_0 = \tau_3 \]
\[ \tau_0 = \tau_0 \rightarrow \tau_3 \]
\[ \tau_0 = (\tau_0 \rightarrow \tau_3) \rightarrow \tau_3 \]
Ex5. Type error: occurs check

• Suppose we want to infer the type of \( f = f \ f \ f \)
Ex5. Type error: occurs check

• How should we prevent our type inference algorithm from looping forever?

• Throw an exception!

  ➤ unify(x, e) should fail if e contains x and e ≠ x

  ➤ E.g., unify(τ₀, τ₀→τ₃) fails!
Type inference by example

1. Basic idea ✓
2. Polymorphism ✓
3. Data types ✓
4. Type error: cannot unify ✓
5. Type error: occurs check ✓
Today

- General discussion of types ✓
- Type inference ✓
- Type polymorphism ✓